國立臺北科技大學
資訊與財金管理系碩士班

碩士學位論文

使用生成式 AI 導入邊緣運算裝置的測試與服務

Introducing testing and services for edge computing devices using generative AI

研究生：鄭亦恩

指導教授：彭祖乙博士

中華民國一百一十四年五月

國立臺北科技大學

資訊與財金管理系碩士班

碩士學位論文

使用生成式 AI 導入邊緣運算裝置的測試與服務

Introducing testing and services for edge computing devices using generative AI

研究生：鄭亦恩

指導教授：彭祖乙博士

中華民國一百一十四年五月

「學位論文口試委員會審定書」掃描檔

審定書填寫方式以系所規定為準，但檢附在電子論文內的掃描檔須具備以下條件：

1. 含指導教授、口試委員及系所主管的完整簽名。
2. 口試委員人數正確，碩士口試委員至少 3 人、博士口試委員至少 5 人。
3. 若此頁有論文題目，題目應和書背、封面、書名頁、摘要頁的題目相符。
4. 此頁有無浮水印皆可。

審定書不用頁碼

摘要

關鍵詞：(請自己填)

摘要為論文或報告的精簡概要, 其目的是透過簡短的敘述使讀者大致瞭解整篇報告的內容。摘要的內容通常須包括問題的描述以及所得到的結果, 但以不超過 500 字或一頁為原則, 且不得有參考文獻或引用圖表等。以中文撰寫之論文除中文摘要外，得於中文摘要後另附英文摘要。標題使用 20pt 粗標楷體並於上、下方各空一行（1.5 倍行高，字型 12pt 空行）後鍵入摘要內容。摘要頁須編頁碼（小寫羅馬數字表示頁碼)。

Abstract

Keyword: (Fill it)

Start writing abstract from here. Start writing abstract from here. Start writing abstract from here. Start writing abstract from here. Start writing abstract from here. Start writing abstract from here. Start writing abstract from here. Start writing abstract from here. Start writing abstract from here.

誌謝

所有對於研究提供協助之人或機構，作者都可在誌謝中表達感謝之意。

目錄

圖目錄

表目錄

第一章　緒論

1.1　軟體開發中的系統測試

近十年間，軟體開發產業正以前所未有的高速發展。[1] 在敏捷開發（Agile）、持續整合與部署（Continuous Integration / Continuous Deployment, CI/CD）技術的推動下，以及透過微服務（Microservices，屬於高度分散式服務範疇）等架構，開發團隊得以實施持續更新（continuous updates），並在極短時間內完成程式修改、測試到上線部署的全流程，從而大幅提升產品迭代頻率與市場回應速度。然而，這種追求開發速度的飛躍，也進一步凸顯了軟體工程中長期存在的一項結構性矛盾（Structural contradiction）：即開發效率的提升與品質保證（Quality Assurance, QA）機制之間的非同步性。在高頻率的版本更新與快速交付環境下，每一次功能增修都可能導致多個服務的重新部署，造成所謂的狀態爆炸（State explosion）」。這種不斷變化的系統狀態，使得經典的驗證與確認（Systematic Verification and Validation, V&V）方法難以執行。儘管系統能夠迅速上線，但傳統測試腳本、驗證邏輯與技術文件往往無法匹配此節奏，造成測試覆蓋率不足、缺陷追蹤效率下降，以及下降文件與實際系統狀態之間的「知識漂移」（Documentation Drift），加劇了系統複雜性與可用能力之間的鴻溝。此即形成了所謂的「速度與品質落差」，開發節奏不斷加快，而品質驗證與知識管理機制卻相對滯後，使得組織在追求開發效率與維持系統可靠性之間陷入結構性取捨。軟體品質不足而導致的產品召回和法律訴訟數量正在迅速增加。軟體驅動的風險和問題（包括缺陷、網路安全攻擊和可用性不足）持續穩步增長。在此背景下，系統測試的目的就是檢查完全整合後的產品是否符合規格要求。對於安全性關鍵系統（safety-critical systems），系統測試必須提供層級化的軟體保證並證明其可信賴性，確保產品在多個維度上的品質，功能性（Functional suitability）、安全性（Security）、可用性（Usability）、可維護性（Maintainability）。由於法律風險和高昂的經濟成本，系統測試的地位已經從低優先級活動轉變為開發流程中不可或缺的核心要素，從需求工程階段就開始整合。在軟體日益複雜的背景下，系統測試是維持系統穩健性（robustness）與營運成功的核心環節，儘管這必須依賴高度自動化來實現。[2]

1.2 RESTful API 語法正確性與語義一致性

軟體開發產業已從早期的單體式應用（Monolithic Architecture）演進至現代的微服務（Microservices）核心組件。[3]REST API 以其輕量、快速且可擴展的特性，成為雲端服務的核心入口。[4] 現代軟體系統普遍透過 HTTP 協議，提供對雲端資源的 CRUD（Create, Read, Update, Delete）操作服務。[5] 由於 REST API 在現代雲端服務中的關鍵作用，確保其品質變得日益重要，其影響服務的可靠性與使用者體驗。而系統測試（System Testing）通常涵蓋驗收測試（Acceptance Testing）是為確保系統功能正確性及符合其功能需求的驗證活動。[6] 然而，隨著微服務與 CI/CD 帶來的快速迭代節奏，傳統的 API 測試方法逐漸暴露出不足之處。

（一）傳統測試的局限與自動化瓶頸

自動化測試在面對敏捷開發及持續整合與部署（CI/CD）所驅動的高速迭代時，仍面臨多重結構性的問題。傳統 API 測試高度依賴專家經驗，透過擷取 API 流量和手動定義的規則來生成測試請求，這對於自動化構成了重大挑戰。[4] 這種方法通常要求測試人員手動為每個非平凡的輸入參數設定有意義的有效測試輸入，即建立和維護資料字典（data dictionaries），因此效率極低。[7] 此外，傳統方式高度依賴專家知識且耗時，透過擷取 API 流量和手動定義的規則來生成測試請求。這對於越來越複雜、參數越來越多的雲端 API 來說，很快就會失效，且大量產生無意義的請求。[4] 為實現 API 測試的自動化，近年來的 RESTful API 自動化測試執行的挑戰和相應技術研究主要採用黑箱測試（Black-box testing），[7] 使用 OpenAPI（Swagger）規格檔作為輸入，透過自動推論 API 之間的依賴關係（例如操作依賴圖，ODG），並依據此依賴關係依序產生測試序列，隨後自動分析回應。然而，在對這些最先進（SOTA）的模糊測試工具（Fuzzers，如 RESTler、Morest、RESTTESTGEN、Miner 等）進行實測後，研究者觀察到兩個普遍且嚴重的重大瓶頸：工具無法產生「語意完整」且「足夠長」的測試序列，[4] 這對全面覆蓋 End-to-end 的驗證構成了挑戰：

1. 因為測試工具用「亂數」或「字典」產生參數，導致生成的輸入數據缺乏語義真實性。雲端服務的 API Gateway 會執行嚴格的語法/語義解析（syntax/semantic check）。一堆不合理的 JSON 結構、型別錯誤、無效 ID 都會立刻被拒絕，造成大

量 400 / 422 錯誤（客戶端錯誤），進而使實際能進到後端邏輯的請求非常少。[4] 這種隨機或結構錯誤的輸入難以有效觸發 API 的核心邏輯。例如，隨機生成方法只能產生 20%的有效 API 呼叫。[7]

2. 現有工具生成的序列通常長度不足，難以觸發雲端服務中難以到達的狀態。這是由於序列生成策略沒有考慮「參數必要性」，也沒辦法產生複雜、跨資源依賴的操作鏈。例如，需要依序執行「建立 group」、「建立 project」、「建立 issue」等複雜操作鏈，若序列只有 1 或 2 步，則根本不可能觸發 API 交互造成的邏輯漏洞。[4]

（二）RESTLess 優化案例

在自動化測試領域中，如何生成同時具備語法正確性（syntactic validity）與語義一致性（semantic coherence）的測試輸入資料，一直是最具代表性的挑戰之一。[7]RESTLess 正是為了解決上述「語義不足」與「序列不夠長」這兩大限制而提出的技術。透過大規模語義增強資料集 RTSet，RESTLess 能夠顯著提升參數合理性，並改善序列推論能力。[4] 本研究將以 RESTLess 為基礎參考，使用 JSON 檔作為輸入規格，結合 LLM 的語意推論能力自動產生更合理的請求參數值與操作序列，並採用基於權重的優化算法，將必要參數視為高權重。透過調整渲染機率，以推論更長、更具多樣性、具跨資源依賴關係的 API 呼叫序列，以作為具備全面覆蓋 End-to-End 驗證自動化測試腳本的基礎生成。[4] 在測試腳本執行後，針對結果進行分析（特別是偵測 50X 錯誤碼），並據以產生測試報告。

1.3 標準化且模組化的 AI Agents 生態協定

在當前軟體產業追求高速疊代，致使 CI/CD pipeline 因傳統測試維護瓶頸而日益承壓之際，人工智慧領域發生了根本性的轉變。[8] 自 Transformer 架構被提出後，為大型語言模型（Large Language Models, LLMs）湧現能力（Emergent Abilities），包括上下文學習（In-context Learning）等特殊能力。[9] LLM 的興起標誌著一個從針對單一任務的「專用 AI」（或傳統機器學習模型）到能夠處理多樣化任務的「通用 AI」（或通用認知引擎，General Cognitive Engine）的典範轉移。LLM 不僅具備強大的語言理解、規劃和推理能力，[10] 還能透過「少樣本提示」（Few-shot Prompting）來處理多樣化的任務。[9]

這種轉變催生了巨大的產業需求，企業意識到 LLM 能夠作為一個通用的認知引擎，用於加速知識工作者的核心任務。[11] 在軟體工程領域，這股需求迅速轉化為利用 LLM 來理解非結構化的技術文件與半結構化的程式碼（如 API 規格），並生成結構化的產出（如測試腳本與技術文件）。[9]

（一）LLM 在系統測試中的需求與自主代理的興起

大型語言模型的出現為解決軟體測試中長期存在的效率與成本問題帶來了突破性的契機。[9]LLM 具備的自然語言理解與程式碼生成能力，使其能夠從 API 規格（如 OpenAPI Specification）中提取結構與語義資訊，[10] 自動產生測試腳本、輸入資料及驗證條件，[9] 並在不同系統模組間建立上下文關聯。[12] 這種能力使測試流程得以在語義層面（semantic level）進行自動化推理與生成，[10] 從而顯著降低人工維護成本與回饋週期。進一步地，隨著資訊技術（IT）自動化的演進，基於 LLM 的自主代理（AI Agents）系統已成為新一代自動化測試與維運的關鍵推手。[12][11] 要理解這一範式轉變，首先必須區分「AI 代理」與「代理式 AI」。

（二）多代理通訊協定的協作

在軟體工程領域，多代理系統被用於自動化軟體測試、故障定位（Fault Localization, FL）[10]、程式碼生成 [13] 以及其他複雜的 IT 運維工作流程。[11] 這些系統能夠分解複雜任務、規劃多步驟流程，並透過工具調用與外部服務互動，實現從測試生成到結果收斂的全流程自動化。然而，當前的 LLM 原生框架（如 LangGraph[8]、LangChain[14] 和 AutoGen[8]）主要透過連續提示鏈（sequential prompt chaining）集中式 LLM 協調來模擬自主性。TB-CSPN 的研究指出，核心問題在於這些架構將語義理解與流程協調混為一談，（conflate semantic processing with orchestration），導致例行性的流程管理任務仍需要昂貴的 LLM 推理，導致例行性的流程管理任務仍需要昂貴的 LLM 推理，限制了可擴展性與真正的自主性。此外，多代理系統的擴展與整合仍面臨嚴重挑戰，例如缺乏標準化的通訊協定導致互通性受限、協作困難（如數據存取不一致和通訊延遲）與安全性風險升高。[8] 為應對上述挑戰，業界開始提出標準化的通訊協議，包括 2024 年 Anthropic 推出模型上下文協定（Model Context Protocol, MCP）、Google 於

2025 年 4 月推出代理人間協定（Agent-to-Agent Protocol, A2A）、同年（2025 年）7 月 IBM 推出 Agent Communication Protocol (ACP) 等，這些協定旨在將分散的 AI 生態系統轉變為穩健、安全且可互通的代理網路。[14] 透過結合 AI 間的協定，能夠建立一個兼具上下文感知調用（context-aware invocation）與去中心化任務協作（decentralized orchestration），分離語義處理（LLM 擅長）和協調邏輯（形式化方法擅長）的生成式 AI 框架。因此，本研究的成果將致力於在軟體工程和 AI 自動化領域中，其具體的技術洞見，實現高效、可驗證且可擴展的代理式 AI 系統，探索相關混合架構（Hybrid Architecture）的可行性與優勢。

1.4 自動化測試生成與驗證流程概述

人工智慧（AI），尤其是大型語言模型（Large Language Models, LLMs）的崛起，為軟體測試的自動化生成與驗證流程帶來了根本性的轉變。基於 Transformer 的 LLM 不僅掌握語言理解，也能進行推理、規劃，使其能夠執行傳統上需要人類智慧的複雜任務。在軟體開發領域中，透過零樣本（Zero-shot）和少樣本（Few-shot）學習能力，[14]LLM 能輕易地從規格文件中提取結構和語義資訊，並轉換為可執行的測試腳本及案例。[5][9] 相關研究已證實此能力：例如 UMTG 透過自然語言處理（NLP）技術從使用案例規範（Use Case Specifications）中萃取物件約束語言（OCL）約束，進而自動生成測試輸入數據及測試腳本（Test Oracle）；[6]KAT 使用 GPT 模型自動生成 RESTful API 的操作依賴圖（ODGs）、測試腳本、約束驗證腳本、測試案例及測試資料；[5]estChain 等多代理框架結合 LLM 與 Python 執行器，以 ReAct 格式的對話鏈中與 Python 直譯器互動，來大幅提高生成測試輸出的準確性。[15]

本研究提出並驗證一套基於 A2A 與 MCP 雙協定整合的創新多代理系統，用於實現高效能的 RESTful API 自動化黑箱測試（Black-box testing）。此架構利用 AutoGen 開源框架的多代理框架 [14] 作為代理間協作的 A2A Host 進行任務協調。首先，測試流程透過系統測試目標 RESTful API 提供的 OpenAPI 標準 JSON 輸入檔開始，[5] 呼叫模型上下文協定（Model Context Protocol, MCP）服務，以整合外部第三方工具和資源程式庫（LangChain），進行 API item 資料處理及分類。LangChain 提供抽象化能力，可建立模組化的工作流程，並具有內建支援向量儲存（vector stores）和檢索器的功能。[14] 為

了加強 LLM 代理的推理品質，本研究也會將手動 SA 文件（manual document），包括操作步驟、測試目的、輸入值、前置條件及預期結果等要素，透過 FastMCP 伺服器以加速上下文傳輸回傳 A2A Host。在系統測試中引入 manual document 作為上下文，可以提供語義知識，類似於 ARTE 透過知識庫（Web of Data）來生成語義有效的測試輸入，從而提高測試腳本準確率。[7]

為了實現複雜的測試任務協作，本研究進一步引入一個多代理（Multi-Agent）系統，包括 Planner Agent（負責任務分解與規劃）[11]、Test Coder Agent（負責生成程式碼腳本）[15]、Executor Agent（負責執行 API 呼叫以及 Analyzer Agent（負責分析執行結果）。這些代理共同進行測試腳本生成及執行測試，並經過多次 re-loop 迭代，[11] 以實現從失敗中學習並持續提升生成品質，回傳成功或失敗的測試結果。最後，在獲得執行結果後，流程再次呼叫 MCP server 的 LangChain 作為 Reporter Agent，整合 API 前處理資訊與執行結果，產生完整的測試報告。[14]

在這個多代理協作過程中，A2A 的作用至關重要。A2A 以標準化的資料結構管理代理之間的狀態傳遞，使任務委派過程具一致性，以實現協作式任務執行。隨後，系統將這些狀態物件完整傳遞至 LangChain Agent，作為上下文提供給 MCP 的工具調用機制。藉由 A2A 與 MCP 的混合架構（Hybrid Architecture），本研究實現了代理間的水平協作溝通（A2A）與模型對工具使用的垂直整合（MCP）的明確分工，有效解決在複雜測試流程中的任務協調成本與上下文一致性問題。此架構的核心價值在於建構一個安全、可擴展且可互通的代理式 AI 生態系統，有效地將 LLM 的語義推理能力與形式化方法的協調優勢結合，以應對現代軟體測試的挑戰。[8]

1.5 研究目標

本研究主要解決當前軟體測試領域在 RESTful API 自動化測試整體的效率問題，提出並驗證一套有效可實際落地的 RESTful API 黑箱測試自動化流程的整合框架，將 A2A 與 MCP 結合，透過混合式多代理架構，明確劃分了代理間的溝通與工具的使用，並在測試時自動化驗證，最終目標在於提升自動化測試的整體正確性，以降低維護成本及回饋週期。

本研究旨在實現以下具體目標：

1. 提出 A2A 結合 MCP 的多協定混合架構：建立一套結合水平代理協作（A2A）與垂直工具整合（MCP）的系統，明確分離語義推理與任務協調，提升代理協作的穩定性與可擴充性。

2. 建構可自動化生成、執行與驗證的 RESTful API 測試流程：利用開放標準（OpenAPI）、文件上下文與多代理分工，形成一套可 re-loop 修正且語義一致的完整測試 Pipeline。

3. 驗證混合式多代理架構在測試準確性的效果驗證：透過實驗與無混合式多代理架構下比較，評估系統在生成測試腳本品質、測試結果一致性與工具協作效率上的優勢。

1.6 研究架構

本研究主要探討如何讓企業在營運產品的開發測試與客戶服務串接，具體為微服務架構系統進行可執行的自動化測試與結果生成，藉由大型語言模型多代理式及 A2A 與 MCP 的混合架構提升測試正確性及一致性。依循論文結構規劃，共分為五章：

第一章為緒論，說明研究背景、問題來源、研究動機，並介紹 AI 代理、多協定混合架構及 RESTful API 測試自動化的相關概念，最後提出研究目標與整體研究架構。

第二章為文獻探討，回顧系統測試、自動化測試、LLM 測試技術、Agentic Workflow、MCP、A2A，以及多代理系統的相關研究，歸納現有方法的限制與研究缺口。

第三章為研究方法，詳細說明本研究提出的混合式代理架構設計，包括系統流程、代理角色分工、A2A 資料結構、MCP 工具調用流程、LLM 推理策略與整體測試自動化方法。

第四章為研究結果與討論，將展示實驗結果，包括測試腳本生成品質、測試成功率、上下文一致性、代理協作成本，以及與現有方法的比較與分析。

第五章為結論與未來展望，將總結本研究實驗成果、技術貢獻及其限制，並提出未來在多代理架構互通的生態下標準化與形式化驗證的混合架構擴展及智能軟體工程應用的可能方向。

第二章　文獻探討

2.1　系統測試的發展歷史

在傳統軟體開發生命週期中，系統測試（System Testing）負責驗證完全整合後的產品是否符合規格要求，是品質保證（Quality Assurance, QA）對照初期需求進行全面驗收的重要階段。然而，隨著對開發速度與彈性的需求不斷提升，特別是在敏捷（Agile）方法興起後，耗時且成本高昂的測試流程（例如大規模回歸測試）逐漸被視為難以持續。組織投入於品質保證與測試的資源已占據 IT 預算的相當比例，使得過去將測試視為獨立階段的做法難以因應當前需求。因此，系統測試逐漸從傳統的後置活動，轉變為嵌入短週期開發衝刺（Sprint）中的持續性驗證與確認（Continuous V&V）流程。這一轉變促使「自動化測試」（Automated Testing）在軟體開發中迅速普及，QA 團隊開始撰寫可重複執行的測試腳本，以確保功能在頻繁修改下仍保持正確性。隨著 CI/CD 的導入，快速部署與「持續測試」（Continuous Testing）成為現代 DevOps 實踐的核心，用以縮短從程式碼提交到上線的整體時間。自動化的系統測試與整合測試被內嵌於 CI/CD 管線（Pipeline）中，使其成為穩定交付品質的必要條件。[2]

- 持續整合（CI）：開發人員頻繁將程式碼合併至中央倉儲，每次合併都會自動觸發建構（Build）與自動化測試（通常涵蓋單元測試與整合測試）。
- 持續部署（CD）：作為 CI 的延伸，當程式碼通過所有自動化測試後，可自動部署至預備環境 (Staging) 或生產環境 (Production)。

自動化測試成為 CI/CD Pipeline 的「品質守門員」（Quality Gatekeeper），讓團隊能夠更有信心地維持高頻率的版本發布。系統測試通常採用「黑箱測試」（Black-box Testing）方法，不要求測試者理解內部程式邏輯，而是透過檢驗輸入與預期輸出之間的一致性來驗證系統行為。[5][16] 於現代 RESTful API 的情境中，多數方法從 API 規範（OpenAPI Specification, OAS）推導測試案例。整體而言，系統測試主要涵蓋兩大類別：

1. 功能測試（Functional Testing）：檢驗系統是否如需求所述運作，確保功能性（Functional suitability）符合明示或隱含的需求。[2]

2. 非功能測試（Non-Functional Testing, NFT）：評估系統執行功能的品質，包括效能效率（Performance efficiency）、安全性（Security）、可用性（Usability）、可維護性（Maintainability）等面向。

儘管自動化測試在 CI/CD 中涵蓋了單元測試、整合測試及部分非功能測試，但仍存在結構性限制。黑箱測試中特別突出的問題是測試腳本（Test Oracle）不足，使得全面自動化難以達成。[17] 傳統自動化測試多依賴 API 回應（例如 HTTP 狀態碼）或 API 規範（如 OAS）的不一致性偵測 [7]，但這不足以確保端到端（End-to-end）的回傳正確性。[16] 在複雜的 RESTful API 系統中，傳統方法難以處理操作間的複雜依賴性（Inter-operation Dependencies）和參數間依賴性（Inter-parameter Dependencies, IPDs）[5][3]，尤其是在通用系統（Ubiquitous Systems）的開發境下，這些挑戰使得測試階段仍需投入大量研究與人力。[18] 微服務架構雖然增強了開發的獨立性與部署的彈性，但也帶來了低可觀察性（low observability）系統行為不穩定性。現有的微服務描述語言（如 OpenAPI Specification, OAS）僅關注單一微服務的 API 描述，卻缺乏對特定業務情境下行為預期或複雜呼叫依賴關係的全面描述，這使得確保規格與實作之間的一致性（Conformance）成為一大挑戰。[19]

綜合上述引用論文的研究，本研究專注於彌補整合性缺口，旨在解決在複雜分佈式環境下，傳統黑箱測試因缺乏語義一致性和長序列推論能力，而難以生成具備高有效性測試案例的問題。為此，本研究專注於為符合 End-to-end 功能需求的系統測試引入高有效性且高穩定性的雙協定自動化測試框架。

2.2 自動化系統測試的挑戰與 RESTful API 測試技術的現況

隨著軟體即服務（Software-as-a-Service, SaaS）成為主流，RESTful API 已成為現代軟體架構的核心，而其品質驗證的重要性愈發凸顯。特別是在安全性關鍵系統（safety-critical systems）中，驗收測試（Acceptance Testing，即系統測試）仍是確保軟體符合功能需求的最終關卡。然而，當開發流程加速朝向敏捷（Agile）與 CI/CD 的短迭代模式前進時，自動化測試面臨的挑戰也變得更加突出。這些挑戰從腳本生成一路

延伸到實際執行，暴露出現行品質保證機制與開發節奏之間根本性的落差。[6] 業界長期面臨的一大瓶頸，是測試腳本難以完全自動化，尤其在缺乏形式化規格或斷言（assertion）的情況下，很難以自動化方式判定系統輸出是否屬於「可接受的正確行為」。[16] 對於複雜功能的 End-to-End 回傳正確性，CI/CD 仍不得不依賴測試人員進行手動檢查，形成實質的人力成本負擔。[6]

2.2.1 腳本撰寫階段的挑戰

在腳本撰寫階段，主要困難來自如何從非結構化需求中生成可執行的測試案例：

1. 傳統自動化方法往往假設系統需求以 UML 行為模型（如活動圖、序列圖）呈現。然而在真實的工業環境中，要建立足夠精確的行為模型，其規格化成本極高，通常不被工程團隊視為可行選項。[6]
2. 大多數需求規格以自然語言（Natural Language, NL）撰寫使用案例規範（use case specifications）。要從這些文件中手動萃取執行場景與輸入資料，不僅昂貴且易出錯。即便使用 NLP 自動生成測試模型，也常需要大量人工修補以補足可執行的輸入資料，導致擴展性（scalability）不足。[6]

2.2.2 自動化測試執行階段的挑戰

即使測試腳本撰寫完成，在執行階段仍會因現代 RESTful API 的複雜性遭遇多重困難：

1. 同時確保輸入資料具有語法正確性（syntactic validity）與語義一致性（semantic coherence）是具挑戰性的問題之一。如 RESTLess 指出許多模糊測試工具依賴字典或隨機渲染參數值，使輸入缺乏語義真實性，導致大量請求在進入系統前便被阻擋。這些無效請求不但造成執行效率低落，也浪費大量資源。[4]
2. 傳統的自動化測試方法，難以有效處理 RESTful API 跨操作（inter-operation）跨參數（inter-parameter）啟發式方法（heuristic approaches）。欄位名稱稍有不一致，就可能誤判或忽略依賴。若改以人工補充依賴（如 RESTest 的方式），則會造成測試工程師沉重的維護負擔。[5]

3. 因無法精確處理複雜依賴關係，多數工具傾向生成短序列、且缺乏參數權重考量，限制了序列類型和長度的多樣性。導致難以觸發隱藏在複雜操作組合下的深層次錯誤（hard-to-reach states），導致測試成效大打折扣。[4]

2.2.3 RESTful API 自動化測試技術的現況

新興的大型語言模型（LLMs）程式碼生成和測試案例生成方面展現出顯著的能力。[6] 於軟體測試領域中，LLM 已被廣泛應用於自動生成測試腳本、約束驗證腳本、測試案例和測試資料，並具備理解與辨識複雜依賴關係的能力。[5] 此外，此外，LLM 也能生成高度語義相關與具真實性的測試輸入值，這對提升 RESTful API 自動化測試的覆蓋率和減少因隨機輸入造成的無效請求具有關鍵作用。[3] 透過利用 LLM 的計算和推理能力，結合外部工具（例如 Python 直譯器，正如 TestChain 框架所示），測試輸出映射複雜的問題可以被拆解並分步推理，從而大幅提高測試案例的準確性，並降低傳統測試腳本帶來的維護負擔。[6]LLM 驅動的測試技術發展意味著 QA/測試工程師的角色將迎來重大轉變，並為自動化系統測試的下一階段演進奠定基礎。[17]

2.3 LLM 驅動的 RESTful API 自動化測試核心方法

LLM 驅動的自動化測試技術，主要在下列三個核心層面解決了 RESTful API 執行測試的挑戰，並提供更全面的功能面 End-to-End 驗證能力 [5]：

1. LLM 的自然語言理解能力，能處理傳統啟發式方法難以捕捉的複雜依賴性。像 KAT 的 AI 驅動方法利用 GPT 模型解析 OAS 文件中的自然語言描述，系統性地提取和辨識操作間依賴性（Inter-operation Dependencies, ODG）與參數間依賴性（Inter-parameter Dependencies, IPD）。LLM 能夠構建操作依賴圖（ODG），明確標示操作之間的順序與依賴關係，並能直接無縫整合到測試腳本生成階段，確保測試序列符合業務邏輯並達成 End-to-End 的正確性驗證。[5]

2. 傳統測試面臨的主要挑戰，是缺乏具備真實意義（realistic）和語義有效（semanti-

cally valid）的測試輸入。[7]LLM 能生成高度語義相關與具真實性的參數值，[15]有效改善這項缺陷，並提升測試覆蓋率並減少無效請求。[7]

早期技術如 ARTE，利用 Web of Data 的知識庫（如 DBpedia）結合 NLP 技術，從 API 規範中自動抽取真實的測試輸入，使 API 呼叫（valid API calls）成功的數量是隨機生成方法的兩倍以上，大幅提高了錯誤偵測能力。[7] 更進一步的技術，如 RESTLess，利用 ChatGPT 執行資料增強，根據歷史有效參數建立大規模高語義的參數值資料集 RTSet，用於補充或替換 OAS 中的參數值，以提高通過語法與語義檢查的成功率。[4]LlamaRestTest 則透過對小型模型（如 Llama3-8B）進行微調（Fine-tuning）量化（Quantization），創建了專門的 LlamaREST-EX 模型，專注於生成特定領域的有效且真實的參數值。[3]

3. LLM 驅動的測試技術不僅在生成階段發揮作用，在實際執行 V&V 流程中，也為傳統的測試腳本問題（Test Oracle Problem）提供了解決方案。雖然 LLM 在複雜計算與嚴格邏輯推理上仍有限制，導致在處理複雜問題時，其生成的測試案例的準確性會急劇下降（主要表現為 Assertion Error，即輸入輸出映射錯誤）。但藉由結合外部工具（如 Python 直譯器），能夠將測試輸出映射的複雜推理任務拆解為多個可控步驟。[15]

   例如 TestChain 的多代理架構中，LLM 能生成程式碼由直譯器執行，逐步完成資料處理與驗證，大幅提高測試案例的準確度。LlamaRestTest 更進一步，成為首個利用 LLM 動態整合伺服器回應的黑箱測試技術。在測試失敗響應（如 4xx）時，它能夠分析伺服器錯誤訊息與參數描述，從而於執行階段即時辨識並精煉及更新參數依賴性 (IPD) 規則和輸入值，使測試流程具備動態調整能力。[3]

2.3.1 本研究採用 RESTLess 之原因

在自動化測試領域中，多篇論文的實驗結果證實了現有工具在語義和序列長度方面的挑戰，一項針對七種最先進模糊測試工具（包括 EvoMaster、RESTler 和 Schemathesis）在 19 個 RESTful API 上的實證比較顯示，這些工具在許多 API 上所實現的程式碼覆蓋率通常低於 50%，這表明存在大量的未覆蓋程式碼。[20] 造成低覆蓋率的主要原因在於，傳統方法生成的序列長度不足，難以觸及複雜、深層次的 API 操

作組合。[21] 此外，由於 OpenAPI 規範本身並不涵蓋參數間的依賴性（Inter-parameter Dependencies, IPDs），且經常存在規格不足（underspecified schemas）語義要求，導致測試序列在第一層輸入驗證就失敗。[22][20] 例如，在某些情境下，API 服務需要由先前請求產生的動態物件（如資源 ID 或檢查碼）作為輸入，純粹的隨機輸入嘗試匹配這些動態物件的機率極低，極可能導致服務僅返回 404 等錯誤狀態碼。[21][22] 雖然 EvoMaster BB 和 Schemathesis 在黑箱模糊測試中表現最佳，但它們的結果變異性仍然很大，且未能完全解決這些深層次的挑戰。[20]

如何生成同時具備語法正確性（syntactic validity）語義一致性（semantic coherence）參數間依賴性（IPDs），[3] 從而難以觸發雲端服務中隱藏的或難以觸及的狀態（hard-to-reach states）。[4] 先前的研究如 ARTE 透過自然語言處理（NLP）技術從知識庫（如 DBpedia）中提取資訊，成功生成了語義有效的測試輸入，其測試參數的現實性（realistic inputs，即語法和語義均有效）平均達 64.9%，顯著優於基線工具 SAIGEN 的 31.8%。[7] 然而，即使是這些語義增強方法，也難以應對現實 API 中複雜的輸入約束和操作序列的依賴關係。[5] 事實上，傳統的模糊測試工具（fuzzers）在實際測試中，通常產生大量缺乏語義意義的無效請求序列，導致難以通過雲端網關的語法和語義檢查，進而無法觸發隱藏在複雜、深層次操作組合中的錯誤。[4]

2.3.2 RESTLess 之研究參考價值

RESTLess 正是為了解決上述「語義不足」與「序列不夠長」這兩大限制而提出的技術。該方法結合了混合優化策略（hybrid optimization strategies），旨在增強現有 REST API 模糊測試的性能。[4]

RESTLess 的主要貢獻包括：

1. 語義增強（Semantic Enhancement）：RESTLess 透過大規模語義增強資料集 RTSet 來提升參數的語義品質。RTSet 包含 62,250 個具備典型 REST API 操作的參數名稱和對應鍵值對，並利用大型語言模型 ChatGPT 進行資料擴增，以產生具有高語義相似性的參數值。這種優化策略能有效地取代或補充原始規範中的參數值，使其生成的測試序列能更順利通過雲端服務的語義檢查。實驗結果顯示，經過 RESTLess 增強後的 SOTA 模糊測試工具，其有效請求序列（Valid sequences，即

20X 狀態碼）的平均增長率達到 42.4%。

2. 渲染順序優化（Rendering Order Optimization）：該模組旨在透過權重式渲染順序優化算法，解決序列長度不足的問題。RESTLess 根據參數的權重屬性（必需參數為高權重，非必需參數為低權重）來優化渲染順序。此策略能夠顯著增加生成的測試序列的類型和長度。這種優化不僅提高了測試效率，讓 RESTLess 在偵測獨特錯誤（unique bugs）方面，平均提升了 54.7%。

綜合以上優勢與應用，本研究將 RESTLess 作為基礎參考，旨在結合 LLM 的語義推論能力與優化算法，克服傳統黑箱測試在處理複雜 API 依賴和長序列生成上的困難。透過利用 LLM 建立 RTSet，有效解決在自動化測試領域中，生成同時具備語法正確性與語義一致性的測試輸入資料，並應用於參數值與操作序列的生成。輸入規格將以 OpenAPI 的 JSON 檔格式，並結合 LLM 的語意推論能力自動產生更合理的請求參數值對 API 規範進行語義增強。基於 RESTLess 的權重式渲染順序優化演算法，以提高推論的有效性，呼叫更具多樣性、具跨資源依賴關係的 API 序列以全面覆蓋 End-to-End 驗證的測試任務。

2.4 大型語言模型領域中 AI Agent 測試技術發展歷程與里程碑

LLM 的出現是重要的里程碑，它將 AI 系統從被動式工具（reactive assistants）轉變為具備自主性（Autonomy）潛力的實體。[23][11] 軟體測試領域迅速利用 LLM 進行創新，特別是在測試用例生成（TCG）和故障定位（Fault Localization, FL）方面。[10][15][11][9][17] 早期的應用包括將 LLM 用於單元測試用例生成、安全測試生成，以及作為生成和突變引擎的通用模糊測試框架（例如 Fuzz4All）。[15][9] 例如，Flakify 這類基於 LLM 的解決方案，可以僅依賴測試案例的原始碼（CodeBERT 模型）來預測不穩定測試（flaky test cases），而不需要存取生產程式碼或定義複雜的特徵集。[13]

為了克服單一 LLM 在面對複雜問題時的計算和推理限制，研究開始結合外部工具。[15] 例如，KAT（Katalon API Testing）利用 GPT 模型和進階提示工程技術來辨識 RESTful API 間的依賴關係，從而提高測試覆蓋率並減少誤報，證明了 LLM 在生成

測試腳本和資料方面的有效性。[5] 在故障定位方面，基於 LLM 的技術如 FlexFL（建構於 Llama3-8B 等開源 LLM）展現出比傳統非 LLM-based FL 技術更高的性能，能夠在 Top-1 排名中定位到 93 個傳統方法錯過的錯誤，並表現出更強的靈活性（Enhanced flexibility），可同時利用錯誤報告和測試套件等資訊。[10]

隨著 LLM 發展成熟，研究轉向利用 LLM 作為核心，建構出具備目標導向自主性（goal-oriented autonomy）極少的人為監督（minimal human oversight）。[11] 人工智慧（AI）領域正朝著高度自主化與協作式的系統演進，由大型語言模型（LLM）AI Agent 與 Agentic AI 的概念，並探討多代理系統在現代軟體開發中的應用與挑戰。AI Agent（AI 代理）自主的軟體實體，它感知環境（例如使用者查詢或感測器資料）來推理並採取行動（例如 API 呼叫、訊息）來完成特定任務或目標。[11] 然而，這個術語常被稀釋，用來描述缺乏自主性、持久記憶或真正協調能力的組件，例如 LangChain 和 AutoGen 某些簡單的流程，或 OpenAI Cookbook 中的硬編碼腳本，常被誤稱為「代理」。[8]

相較之下，Agentic AI（代理式 AI）多代理系統（Multi-Agent Systems, MAS），其中多個專業代理協作、協調並規劃，以實現複雜、高層次的目標。Agentic AI 是一種自主的、目標驅動的系統，專注於自治（Autonomy）、適應性（Adaptivity）和目標驅動的推理（Goal-driven Reasoning），能夠在極少人類監督下，長時間獨立運作。[11] 模組化多代理系統透過明確定義的協議進行協調，能夠執行複雜的、多步驟的任務。[8]

2.5 多代理系統與傳統架構的挑戰

多代理系統在系統開發中的應用極為關鍵，被用於自動化軟體測試、故障定位（FL）、IT 運維工作流程以及其他複雜的任務。[8]MAS 能夠將複雜的任務分解成多個子任務、規劃多步驟流程，並透過工具調用與外部服務互動，實現從測試生成到結果收斂的全流程自動化。[11] 儘管 LLM 原生框架如 LangChain、AutoGen 和 LlamaIndex（前身為 GPT Index）等框架已經出現，旨在簡化代理開發，但它們在實現大規模協作時暴露出嚴重的架構缺陷。

目前，業界已有多個 LLM 驅動工作流程的主流框架嘗試實現大規模協作與自主性，其中最具代表性的包括 LangChain、LlamaIndex 和 AutoGen。這些第一代 LLM 原

生框架雖然展現了巨大的潛力，但同時也暴露出固有的架構限制，這些限制阻礙了真正可擴展、可驗證的多代理系統的發展。[8]

1. LangChain（PaaS/Library）：

    - 優勢：提供了強大的抽象化能力（例如「鏈」抽象化，Chains），用於在模組化工作流程中串聯 LLM 呼叫、記憶緩衝區和函數調用。它內建支援檢索器、向量儲存（vector stores）檢索增強生成（RAG）上下文推理（contextual reasoning）結構化、目標導向的工作流程。[14]LangChain 的衍生框架 LangGraph 引入了持久記憶、狀態轉換和模組化任務節點，將工作流程作為有限狀態機或圖形執行。LangChain 的衍生框架 LangGraph（於 2025 年推出）更進一步引入了持久記憶、狀態轉換和模組化任務節點，將工作流程作為有限狀態機或圖形執行。[8]
    - 限制：其架構將語義理解與流程協調混為一談（conflate semantic understanding with process orchestration）。這種 LLM 介導的協調要求在每一個協調決策點都進行 LLM 推理，導致例行性的流程管理任務產生昂貴的語言模型呼叫。這種集中式的協調方式建立了可擴展性的瓶頸，[8] 並且其工作流程不能被正式驗證（formally verified）不原生包含自我反思或優化功能，其智能主要由開發者指導。[11]

2. LlamaIndex（Data Framework）：

    - 優勢：旨在將 LLM 與客製化知識庫整合，連接 LLM 至使用者擁有的知識。該框架提供文件載入器、索引包裝器和一個「工具註冊表」（tool registry），能將使用者查詢映射到 API 呼叫。[14]
    - 限制：其核心設計更偏向「知識庫」，而非「任務執行與協調」。該框架專注於將 LLM 與外部知識整合，例如透過文件載入器和索引包裝器來實現。

3. AutoGen（Agent Framework）：

    - 優勢：專為多代理系統設計，透過會話互動（conversational interactions）協作與協調。[8] 該框架支援規劃、反思（reflection）和優化，使代理能夠評估

計劃並改進其行動，透過結合工具使用和團隊合作來處理複雜的、分佈式的目標，從而減少錯誤並提升性能。[11]

- 限制：依賴「基於提示的協調」（prompt-based coordination），而非正式的多代理協議。這導致了脆弱的溝通模式（brittle communication patterns），且缺乏持久的代理狀態（persistent agent state）來超越對話歷史，也沒有機制進行長遠規劃或目標修正。因此，AutoGen 也是透過連續提示鏈和集中式 LLM 協調來模擬自主性。[8]

這種主要透過連續提示鏈（sequential prompt chaining）集中式 LLM 介導的協調（LLM-mediated coordination）來模擬自主性，導致例行性的流程管理任務仍需要昂貴的 LLM 推理，嚴重限制了效能和可擴展性。作為對比，TB-CSPN（Topic-Based Communication Space Petri Net）等混合架構，透過將語義處理（LLM 擅長）與協調邏輯（Petri Net 擅長）分開，展現了顯著的優勢。實證評估顯示，TB-CSPN 比 LangGraph 風格的協調快 62.5%，LLM API 呼叫減少 66.7%，證明了架構分離的效率。此外，多代理系統的擴展與整合仍面臨嚴重挑戰，例如缺乏標準化的通訊協定導致互通性受限、協作困難（如資料存取不一致和通訊延遲）與安全性風險升高。[8]

本研究所提出的自動化測試框架將綜合 LangChain 的程式庫和 AutoGen 的 Agent Framework 的優勢作整合。本研究採用 AutoGen 框架作為協調基石，引入 Planner Agent（策劃與協調）、Test Coder Agent（測試腳本生成）、Executor Agent（腳本執行）和 Analyzer Agent（結果分析與比較）四大關鍵角色。這些 Agent 將透過標準化的資料結構進行狀態傳遞，確保任務在不同階段無縫交接。LangChain 將作為專業工具庫（Programmatic Libraries）前置處理與分類服務，以及結合前置處理的分類上下文來生成詳盡且結構化的測試報告。本研究的整合架構，透過 AutoGen 實現了流程的靈活性與協調性，同時透過 LangChain 實現了工具使用的準確性與專業性。這種結合不僅為 API 測試自動化提供了一個強大、可重複驗證的框架，也為未來更複雜的多領域、跨協議 Agent 協作研究奠定了堅實的基礎。

2.6 Multi-Agent 自動生成測試案例與 Hybrid Architecture 的協作

2.6.1 Multi-Agent 自動化生成測試案例

LLM 驅動的多代理系統 (Multi-Agent Systems, MAS) 正成為自動化測試案例生成領域的重大前沿，旨在克服單一 LLM 或傳統測試框架在處理複雜任務（特別是涉及規劃和高精度計算）時的固有限制。多代理協作利用分工（specialization）結構化通訊（structured communication）、形式化協調（formal coordination），大幅提升了測試用例的準確性、多樣性及自動化效率。[15][11][23][8]

多代理系統的核心優勢在於將複雜的測試生成流程分解為由專業代理人處理的子任務，從而增強了整體效能及自動化。TestChain 框架是一個顯著的里程碑，它將測試案例生成任務分解為兩個連續子任務，測試輸入生成（Test Input Generation）和測試輸出生成（Test Output Generation）。Designer agent 專注於生成多樣化的測試輸入，包括基礎和邊緣案例，Calculator agent 則負責確定對應的預期測試輸出，並透過 ReAct 格式的對話鏈與 Python 解譯器進行互動。以 GPT-4 作為基礎模型的 TestChain 在 LeetCode-hard 資料集上的測試案例準確性（Accuracy）提高了 13.84%，證明了這種分工和外部工具協作的有效性，亦顯著減少了輸入及輸出映射中的複雜性和不準確性。[15] AutoGen 等框架專為多代理應用設計，透過會話互動（conversational interactions）實現代理間的自主協作與協調。AutoGen 允許代理進行規劃（planning）和反思（reflection），並透過團隊合作來進行流程管理，處理複雜的、分佈式的目標，從而減少錯誤並提升性能。MetaGPT 是一個更進階的多代理協作框架，它將複雜問題分解為子問題，並自動根據代理的專業分工分配角色，協調代理的互動來解決整體問題。這種有組織的「代理人社會」（society of agents）能夠提高效率和可擴展性。[11]

在微服務架構複雜的 End-to-End 系統中，多代理協作變得尤為重要。微服務系統由多個獨立開發、部署、升級和擴展的小型服務組成。[19] 多代理框架（如 TB-CSPN）的出現，旨在克服第一代 LLM 框架（如 LangGraph 和 AutoGen）在流程協調上的限制，這些限制源於它們將語義理解與流程協調混為一談。TB-CSPN 透過架構分離將

LLM 限制於語義處理（如主題提取），而將協調邏輯委託給形式化方法（如彩色 Petri 網，Colored Petri Nets, CPNs），從而能以更高的效率和可驗證性來管理多代理協作，確保代理間的通訊可追蹤、可解釋且可靠。TB-CSPN 透過主題標籤令牌（topic-tagged tokens）實現代理間的結構化通訊，支援顧問（Consultant）、主管（Supervisor）和工人（Worker）三層代理體系，這為在複雜、多步驟環境中自動生成和執行測試序列提供了嚴謹的基礎。[8]

綜合以上觀點，多代理系統透過分工任務、利用外部工具進行精確計算和形式化協調，已成為提升自動化測試案例生成品質和效率的關鍵技術。這種協作模式特別有利於解決單一 LLM 在長序列推理和高精度計算方面的挑戰，是未來複雜軟體系統（如微服務、企業工作流程）自動化測試的核心方向。

2.6.2 Hybrid Architecture 協作機制的發展與創新理念

（一）AI Agent 標準化協定概述與比較

傳統上，代理系統之間採用的臨時性整合（Ad-hoc integrations）難以大規模化、確保安全性和跨領域泛化。為了解決多代理系統在擴展和整合時遇到的挑戰（例如缺乏標準化的通訊協定、協作困難及安全性風險升高），業界各大廠針對不同的互通性層級開始推出標準化的通訊協定，旨在將分散式碎片化（fragmented）AI 生態系統轉變為穩健、安全且可互通的代理網路。[14]

1. Model Context Protocol（MCP）：由Anthropic 於 2024 年推出，提供一個客戶端-伺服器介面（JSON-RPC），用於安全上下文攝取（secure context ingestion）結構化工具調用（structured tool invocation）。它旨在透過提供一個通用、與模型無關（model-agnostic）的介面，來標準化應用程式如何向 LLM 傳遞工具、資料集和採樣指令，解決 LLM 缺乏上下文標準化的問題。

2. Agent-to-Agent Protocol（A2A）：Google 於 2025 年 4 月推出，旨在實現 AI 代理間的安全、結構化、可互通的協作（interoperable collaboration），特別適用於企業代理工作流程（enterprise agent workflows）。它支援點對點（peer-to-peer like）任務委派，並使用 Agent Cards 來描述代理的能力和安全調用方式。

3. Agent Communication Protocol（ACP）：於2025 年 7 月由 IBM 推出，是一個通用協定（general-purpose protocol），使用 RESTful HTTP 介面，支援 MIME-typed multipart messages 以及同步和非同步互動。它設計輕量且運行時獨立（runtime-independent），適用於可擴展的系統整合。

4. Agent Network Protocol （ANP）：ANP 是一個開源通訊框架，它採用去中心化點對點（Decentralized Peer-to-Peer, P2P）模型，旨在實現開放網路（open internet）跨平台代理協作。它利用 W3C 去中心化識別碼（DIDs）和 JSON-LD graphs 進行身份驗證和語義交換。

下表比較了 MCP、ACP、A2A 和 ANP 這四種協定在分散式環境下，關於互通性、安全性和擴展性的關鍵特點：[14]

表 2.1: MCP, A2A, ACP, ANP Protocol 模型架構、特點與應用範疇比較表

Protocol	Module	Scopes	Interoperability	Security	Scalability
MCP	客戶端-伺服器 (JSON-RPC)	垂直整合：LLM↔外部工具/資源整合。標準化上下文攝取和結構化工具調用。	高（LLM 與工具整合）。	Token-based auth, 需透過 mTLS、OAuth 2.1+ PKCE 緩解工具投毒和憑證竊取。	集中式伺服器假設，資源由客戶端管理。
A2A	類點對點 (Client↔Remote Agent)	企業協作：實現代理間的結構化互動與任務委派。使用 Agent Card 實現能力發現。	適合企業信任邊界內的協作。	DID-based 身份驗證，使用 JSON Web Signatures (JWS) 防止任務偽造。	支援 SSE/Push Notifications，適合分佈式生態系統。
ACP	代理中介客戶端-伺服器 (RESTful HTTP)	基礎設施層：通用通訊協定，支援多模態訊息和可擴展系統整合。	模型無關 (Model-Agnostic)，採用 RESTful HTTP，適合廣泛部署。	Bearer tokens, mutual TLS，JWS。	輕量且運行時獨立，支援可擴展的代理調用。
ANP	去中心化點對點 (P2P)	開放網路互聯：實現跨平臺、去中心化的代理發現和安全協作。	最強（開放網路），使用 JSON-LD 進行語義資料交換。	依賴 W3C DID 進行去中心化身份驗證。	適用於開放代理市場，但協商開銷高。

（二）雙協定協作的創新案例與優勢

在多代理系統的發展中，單一協議難以解決所有問題，因此整合多個協議成為實現高效能自主系統的關鍵。針對不同的場景各個協定各有互通性優勢，MCP 解決 LLM 與工具的垂直整合問題，確保上下文一致性，而 A2A 和 ACP 則側重於代理間的水平協作與通訊，提供結構化訊息傳遞和任務委派。例如，在 IT 事件響應案例中，Incident Coordinator Agent（事件協調代理）主要使用 A2A 協議來委派任務和追蹤狀態。相對地，MCP 協議作為上下文感知層（contextual awareness layer），負責垂直整合。它使得

代理能夠以一致且安全的方式存取診斷工具、知識庫和系統監控器。例如，Diagnostic Agent（診斷代理）利用 MCP 存取診斷工具，同時確保在複雜的多步驟操作中上下文完整性（context integrity）得以維持。[24] 在實踐中，MCP 與 A2A 雙協定的協作整合展現了顯著的創新案例與優勢，特別是在複雜的 IT 事件響應等領域，能夠克服傳統多代理系統在上下文分享、能力發現和訊息格式不相容等方面的限制。[24] Vaibhav Tupe 和 Shrinath Thube 於 2025 年在研討會中發表一篇《Demonstrating Multi-Agent Collaboration via Agent-to-Agent and Model Context Protocols: An IT Incident Response Case Study》，展示了一個整合 Google 的 A2A 協定和 Anthropic 的 MCP 協定的多代理系統。在這個 IT 事件響應案例中，系統由三種專業代理人組成，

- 事件協調代理（Incident Coordinator Agent），主要使用 A2A 協定，負責委派任務和追蹤系統範圍內的狀態；
- 診斷代理（Diagnostic Agent），利用 MCP 協定作為上下文感知層存取診斷工具，同時確保在複雜的多步驟操作中上下文完整性（context integrity）得以維持及
- 解決代理（Resolution Agent），依賴 MCP 進行部署和配置工具的安全存取，同時使用 A2A 向協調代理報告解決狀態。

1. MCP 提供了一個 JSON-RPC 客戶端-伺服器介面，用於安全且結構化的工具調用和上下文攝取（context ingestion）。這確保了 LLM 可以一致地存取外部資料和工具。

2. A2A 旨在透過標準化的通訊層實現 AI 代理間的結構化互動和任務委派。將 MCP 與 A2A 整合，結合了 A2A 的結構化訊息傳遞和能力發現與 MCP 的上下文感知機制。

3. 整合創造了一個統一的框架，使具有不同專業的代理能夠無縫協作，同時保持上下文一致性。例如，在 IT 事件響應案例中，協調代理可以使用 A2A 進行任務委派，而診斷代理則可以利用 MCP 存取具備正確上下文的診斷工具。這種雙協議整合解決了跨協議通訊中的能力表徵、上下文轉換和訊息路由等關鍵挑戰，使得代理能夠在維護共享理解的同時，跨協議邊界進行協作。

MCP 與 A2A 混合架構（Hybrid Architecture）結合了 A2A 的結構化訊息傳遞和能力發現，以及 MCP 的上下文感知機制，它使得 LLM 能夠以一致且安全的方式存取外部資料和工具，為可擴展、可互通、且安全的多代理生態系統奠定基礎。這種雙協議整合解決了跨協議通訊中的能力表徵、上下文轉換和訊息路由等關鍵挑戰。整合層在協議格式之間進行轉換，確保在一個協議中建立的上下文在透過另一個協議存取時仍保持一致。然而，有研究也警告，這類協議的組合可能會引入語義不匹配、安全性風險和協調複雜性，需要像 TB-CSPN 這種形式化混合架構來策略性地嵌入 AI 組件，以提供協調保證和更高的效率。[8] 因此，為解決前述研究所指的挑戰，本研究之自動化測試腳本生成預處理將以 RESTLess 為基礎。

基於上述研究，本研究提出的自動化測試框架將綜合 LangChain 的程式庫及 AutoGen 的 Multi-Agent Framework 的框架，結合 MCP 與 A2A 雙協定的混合架構下的框架。採用 AutoGen 框架作為代理間協作的 A2A 主機（A2A Host）。AutoGen 透過會話互動（conversational interactions）基於提示的協調來模擬自主性。雖然 AutoGen 依賴提示式協調而非形式化協議，但它在本研究中提供了流程的靈活性與協調性，體現了 A2A 在結構化互動和任務委派上的作用。同時利用 LangChain 程式庫，透過 FastMCP 伺服器實現工具服務的 MCP 專業能力。LangChain 提供強大的程式庫和抽象化能力來串聯 LLM 呼叫和函數調用，這使其成為實現 MCP 垂直整合和上下文感知工具調用的理想選擇。

這種設計明確劃分了代理間的溝通與工具的使用，從而解決了複雜測試流程中任務協調和上下文一致性的挑戰。透過 AutoGen 的協作（類 A2A）實現了流程的靈活性，同時透過 LangChain 的工具呼叫（類 MCP）實現了工具使用的準確性與專業性。這種雙協議的結合，為 API 測試自動化提供了一個強大、可重複驗證的框架，也為未來更複雜的多領域、跨協議 Agent 協作研究奠定了堅實的基礎。

第三章　研究方法

3.1　研究架構與整體流程

本章節根據文獻探討對於 RESTLess 混合優化策略，與本研究提出的自動化測試框架採用創新的分散式雙主機架構（Distributed Dual-Host Architecture）融合，其核心優勢在於實現責任隔離與協議互操作性。基於對 A2A 協定和 MCP 協定的雙協定整合的流程設計，RESTLess 的兩大優化策略將分別應用於不同的代理人（Agent）和階段，以最大化 LLM 的專業分工和雙協定（A2A 與 MCP）的優勢，旨在克服傳統單一 LLM 代理框架在上下文一致性和跨主機任務協調方面的挑戰。整體流程圍繞五大關鍵階段展開，從接收測試需求開始，透過代理間的結構化通訊和專業工具整合，最終輸出完整的測試報告，並自動化銜接 CI/CD 流程中的 Git 操作。首先，由 Planner Agent 啟動流程，將 OpenAPI JSON 輸入檔及 Manual Document，透過呼叫 FastMCP 服務作為外部工具知識庫進行執行資料處理及垂直整合；接著，針對回傳的測試條件後，由 Planner Agent 使用 A2A 協定的任務委派模式，將生成測試腳本的任務傳遞給 Test Coder Agent 及 Executor Agent 進行腳本生成與執行；第三階段為結果分析與自校準，Executor Agent 將原始的執行結果（包括 HTTP 狀態碼狀態碼 2XX/4XX/5XX），透過 A2A 協定回傳給 Analyzer Agent，若失敗時會啟動 Re-loop 流程，將失敗原因和當前狀態，回饋給 Planner Agent，並指派 Test Coder Agent 進行修復性代碼生成；最後，進而再次呼叫 FastMCP 服務，啟動 Reporter Agent 執行外部操作（GitOps），完成最終報告生成與結案，並發送 A2A 請求給位於 Host 2 上的 GitOps Agent，允許代理跨越協議邊界進行協作。

3.1.1　主機劃分與角色分配

本研究提出了一個高度模組化且具備跨系統協作能力的自動化測試生成與驗證框架。此框架的核心理念是結合 A2A 協定實現代理間的任務協調與狀態傳遞，並透過 MCP 協定實現上下文感知的專業工具服務。整個系統由兩個邏輯隔離的主機和六個專業代理共同驅動。本架構由兩組獨立的主機環境組成，整體流程涵蓋四個主

要大角色與協定任務：AutoGen Host、FastMCP Client、FastMCP Server、External A2A Host、GitOps Host，以及六個代理 Agents 與子任務: Planner Agent、Test Coder Agent、Executor Agent、Analyzer Agent、Reporter Agent、GitOps Agent，，以達成功能隔離和安全分權，以下為詳細分權及說明：

1. Host 1：AutoGen A2A Host (測試與分析主機)
    - 角色定位：作為主要的 A2A 控制中心和 FastMCP Client，負責整個測試流程的生命週期管理。
    - 組成與任務：包含 Planner Agent、Test Coder Agent、Executor Agent、Analyzer Agent 四個核心 Agent。它也是 FastMCP Server 的物理部署位置，並在內部以 LangChain 庫的形式提供專業的 MCP 工具服務（例如：API Item 前置處理、報告生成）。
    - 協議職責：主導 Host 1 內部 Agent 間的 A2A 協作，並作為 A2A Client 向 Host 2 的 External A2A Host 發送跨主機請求。
2. Host 2：External A2A Host / GitOps Host (部署與操作主機)
    - 角色定位：作為 A2A Server，專門接收來自 Host 1 的高權限操作請求，實現安全隔離。
    - 組成與任務：僅包含 GitOps Agent。該 Agent 負責對 SUT (System Under Test) 的 Git Repository 執行所有 Git 操作，例如版本標記（Tag）、分支合併（Merge）等。
    - 協議職責：透過 A2A 介面接收 Host 1 的任務，執行 GitOps 流程，並將操作的成功或失敗狀態回傳給 Host 1，確保了測試與部署環境的權責分離。

3.1.2 自動化測試生成與驗證流程概述

整個自動化流程是一套由 A2A 協議驅動的閉環（Closed-Loop）協作，其中 MCP 協議在關鍵節點提供專業能力與上下文一致性。各階段間之互動說明如下：

階段（一）輸入與前置處理（Input & Pre-processing）

本階段將以接收使用者提供的 OpenAPI JSON 檔案（定義測試目標）與 Manual Document（包含預期結果及操作步驟等，作為高語義測試案例）。整體服務流程由 Planner Agent 啟動，透過呼叫 FastMCP（由 LangChain 程式庫支持的工具互動層），執行 API item 的資料處理與分類。此 MCP 服務中將嵌入 LLM-Based Specification Semantic Optimization 模組（類似 RESTLess 的語義增強演算法）。處理後的分類結果和 Manual Document（包含人類預期結果）一同被封裝成核心上下文（Core Context），作為 A2A 狀態資料結構的一部分，注入到測試流程中。

階段（二）腳本生成與執行（Code Generation & Execution）

接繼使用 A2A 協定的任務委派模式，將生成測試腳本的任務傳遞給 Test Coder Agent，依據 A2A 狀態資料中包含的 MCP Context（即語義增強後的規格），實施 Weight-Based Render Order Optimization 策略（類似 RESTLess 的渲染順序優化）。然後，透過機率 ω 調整來決定是否添加低權重（Non-Required）參數進行測試，從而生成更長、更多樣化的測試序列，以檢測隱藏在複雜操作組合中的錯誤。腳本隨後被傳遞給 Executor Agent，執行針對測試目標進行 RESTful API 的黑箱測試。

階段（三）結果分析與自校準（Analysis & Self-Correction）

在此階段，Executor Agent 及 Analyzer Agent 扮演自動化測試流程驗證與修正的重要橋樑，流程如下：

1. Executor Agent 將原始的執行結果（包括 HTTP 狀態碼 20X/50X）透過 A2A 協定回傳給 Analyzer Agent。

2. Analyzer Agent 利用 Manual Document 提供的預期結果進行結果比對（即測試案例），判斷測試成功或失敗。

    - Re-loop 機制（自校準）：若失敗，Analyzer Agent 將失敗原因和當前狀態（維持在 MCP Context 中）回饋給 Planner Agent。Planner Agent 啟動 Re-loop 流程，指派 Test Coder Agent 進行修復性代碼生成，直至測試成功。

階段（四）CI/CD 外部操作（GitOps Integration）

階段（三），一旦測試成功，Executor Agent（類似於 IT 事件響應系統中報告解決狀態的代理）發送 A2A 請求給 GitOps Host 的 GitOps Agent。此實施展現了雙協定的互通性，允許代理跨越協議邊界進行協作。GitOps Agent 接收 A2A 結構化請求後，執行相關的 Git 操作（例如打 Tag、Merge Branch 或 Pull Request），實現從測試到部署的自動化銜接。

階段（五）報告生成與結案（Reporting & Closure）

最後，Analyzer Agent 回傳資料給 Planner Agent 後將再次呼叫 FastMCP 服務，啟動 Reporter Agent。透過 MCP 協定的上下文感知層，存取並整合整個 A2A 流程的所有日誌、執行結果、Manual Document 的預期值，以及前置處理的分類結果。MCP 在此階段確保了所有工具和代理在整個複雜流程中上下文的一致性，最終生成一份完整且結構化的最終測試報告。

3.2 前置處理的理論與設計

為建構具備高語義品質的核心上下文（Core Context），此上下文對於後續代理（Agent）生成有效且能通過雲端閘道檢查的測試腳本至關重要。本階段的設計核心，在於整合模型上下文協議（MCP）的垂直整合優勢，以及源於 RESTLess 框架的語義優化策略。

此階段的理論基礎聚焦於克服僅依賴原始 OpenAPI 規範作為測試輸入的侷限性。傳統的 REST API 模糊測試（Fuzzing）方法，其挑戰在於生成的請求序列普遍缺乏業務語義價值，從而導致大量的語義缺失請求在雲端閘道層被語法或業務邏輯檢查所阻擋，嚴重影響測試效率。為解決此問題，本研究引入 MCP 作為上下文感知層（Contextual Awareness Layer），以確保跨 Agent 與工具間的語義一致性。Planner Agent 作為 FastMCP Client，透過呼叫 FastMCP Server（其工具互動層由 LangChain 程式庫提供支持），實現對規格數據和人類知識的安全且一致性存取。此外，我們整合了基於 LLM 的規格語義優化模組（LLM-Based Specification Semantic Optimization），此模組旨

在增強原始 OpenAPI 規範的語義豐富度，進而提升有效請求序列的通過率。

本階段的實施流程旨在將非結構化或半結構化輸入轉化為高語義的結構化上下文，其步驟如下：系統首先接收兩類輸入：OpenAPI JSON 檔案（定義 API 結構與參數名稱）和 Manual Document（包含高語義測試案例、預期結果及操作步驟等）。由 Planner Agent 隨即透過 A2A 協定發出任務，並作為 FastMCP Client 呼叫 FastMCP 服務，以執行 API item 的資料處理與分類。在 FastMCP 服務內部，嵌入的 LLM-Based Semantic Optimization 模組執行語義增強演算法：

1. 模組透過精確的 Prompt Engineering，指示底層 LLM 學習原始規範中參數的潛在語義（例如，從名稱判斷用途），並生成多個語義相似且具代表性的參數值集合（RTSet）。
2. 緊接著，LLM 自動掃描原始 OpenAPI 規範，將生成的 para-value 鍵值對從 RTSet 替換或補充到對應參數的 enum 字段中。此步驟旨在確保 Test Coder Agent 能夠存取到具備業務語義的有效輸入值，而非隨機生成的無效數據。

經過 FastMCP 服務處理後，最終的輸出結果（包括分類結果、語義增強後的規格，以及 Manual Document 中包含的人類預期結果）將被標準化封裝成核心上下文（Core Context）。此核心上下文作為 A2A 狀態資料結構的一部分，被注入到測試流程中。此機制確保了後續的 Test Coder Agent 在生成腳本時，能夠在語法和語義上都依據一個一致且強化過的測試依據，從而大幅提升測試腳本的準確率和有效性。

本研究建議利用 LangChain 程式庫作為 FastMCP 服務中的底層工具互動層。LangChain 的 Tool 抽象機制用於將「OpenAPI 語義增強」定義為一個可供 Agent 安全呼叫的服務。該 Tool 內部利用 LangChain 的 LLM Chain 執行所有 Prompt Engineering 和數據擴增邏輯。FastMCP 服務在此充當上下文管理器，負責將 LLM 生成的高語義參數值安全地寫入到核心上下文中，確保 Planner Agent 透過 MCP 呼叫此服務時，能夠獲得上下文完整性的保證。

這個強化後的上下文隨後被注入到 A2A 流程中，供 Test Coder Agent 在階段二進行腳本生成時使用，確保生成的測試請求在語法和語義上都具有高通過率。

表 3.1: 語義優化與上下文建立範例設計對照表

Step	Content	Example
輸入：原始規格	OpenAPI 規範：id 參數的類型為 `string`，但無 `enum` 字段。	缺乏語義，可能被 Fuzzer 隨機填入 “123” 或 “abc”，難以通過語義檢查。
輸入：Manual Document	預期結果：“系統應接受格式為 U_XXXXXX (X 為數字) 的使用者 ID。”	提供了高語義的有效值格式。
MCP 服務執行語義優化	Planner Agent 呼叫 FastMCP 服務執行 LLM 數據擴增（類似 RESTLess）。	透過 Prompt 告訴 LLM id 的語義應是 U_XXXXXX 格式。LLM 生成 RTSet 範例：[“U_001234”, “U_987654”, “U_555555”]。
生成語義增強規格	FastMCP 服務將 RTSet 寫入規格的 `enum` 字段。	新的 OpenAPI 規範：id 參數新增 enum:[“U_001234”, “U_987654”, “U_555555”]。
上下文封裝	Planner Agent 將增強後的規格和 Manual Document 封裝。	A2A 狀態資料結構中包含：{API_SPEC_ENHANCED:< 增強後規格 >, EXPECTED_RESULT:“系統應接受...”,...}。

3.3 腳本生成與執行方法設計

此階段的設計依託於多代理系統的結構化任務委派能力，並融合了參照 RESTLess 框架的混合語義優化策略，旨在生成具備高效率和高覆蓋率的黑箱測試案例。將第一階段建構的語義強化上下文（MCP Context）轉化為可執行的測試序列的關鍵環節。主要奠基於 A2A 協議的水平協作特性，以及 LLM 驅動的語義優化，以突破傳統測試工具在處理 RESTful API 間複雜操作依賴和長序列生成上的技術瓶頸。

3.3.1 測試腳本生成

流程始於 Planner Agent 透過 A2A 協定（由 Autogen 框架支持）啟動任務委託模式，將測試生成的任務與完整的 A2A 狀態資料（包含來自 MCP 服務的高語義化上下文）傳遞給 Test Coder Agent。A2A 協定在此提供了標準化的訊息傳遞框架，確保了任務目標、上下文資訊和依賴圖（ODG）要求的清晰、無損傳遞，為腳本生成提供了穩定的輸入依據。

Test Coder Agent 的生成是基於 RESTLess 的權重參數渲染順序優化策略 (Weight-Based Render Order Optimization)，此策略旨在平衡測試序列的有效性（Validity）與多樣性（Diversity）。該策略首先根據 API 規範，將參數區分為高權重參數（例如 Required 必填欄位）和低權重參數（例如 Non-required 可選欄位）。高權重參數被視為確保請求

序列符合語法與業務邏輯、能夠通過雲端閘道檢查的關鍵要素。優先使用高權重參數和階段一語義增強後的 RTSet 資料進行渲染，以確保生成的初始測試序列 S_1 具備高成功率。同時，Agent 必須嚴格遵循操作依賴圖（ODG）的順序要求，確保前置操作的輸出（例如 Session Token 或 ID）被正確擷取並注入為後續操作的輸入。在完成有效序列生成後，Test Coder Agent 根據可靈活調整的機率 ω（例如預設值 $\omega = 80\%$），決定是否將低權重參數納入測試序列。此機率抽樣機制有效地擴展了測試案例的多樣性，增加了發現潛在邊緣錯誤（Edge Cases）的可能性，同時保持了對語義有效性的高度關注。

3.3.2 腳本執行與黑箱測試驗證

生成的測試腳本（例如 Python 腳本）隨後被傳遞給 Executor Agent 執行。Executor Agent 執行針對測試目標 RESTful API 的黑箱測試。其操作不依賴服務的內部原始碼，而是完全根據語義增強後的 OpenAPI 規範和 Manual Document 中的預期輸出來驗證系統的外部行為。Executor Agent 負責實際的 HTTP 請求發送，並準確記錄所有執行細節和原始結果（Raw Result），包括但不限於 HTTP 狀態碼（2XX, 4XX, 5XX）和回應主體。這些原始結果隨後透過 A2A 協定傳遞給下一階段的 Analyzer Agent，作為結果分析與自校準的基礎輸入。

透過上述設計，本階段確保了測試腳本不僅具備 LLM 的靈活性，更結合了工程的嚴謹性，為後續的自動化驗證與 CI/CD 流程奠定了高效且可靠的基礎。

3.4 結果分析與自校準驗證機制設計

為應對整個自動化測試流程中驗證（Verification）回復（Recovery）錯誤復原（Error Recovery）結果分析與自校準驗證機制，此階段結合了 A2A 協定的訊息交換（Message Exchange）MCP Context 所提供的上下文一致性，以實現高度自主的測試流程。本研究於此階段承繼 Executor Agent 的原始執行結果（即測試執行過程中產生的 Artifacts 透過 A2A 協定回傳給 Analyzer Agent，包括 HTTP 狀態碼）判斷測試成功或失敗，決定是否啟動 Re-loop 機制。

當Analyzer Agent 判斷測試失敗後，它會將結構化（例如 JSON 或 Topic-tagged tokens）的失敗原因和當前狀態透過 A2A 協定回饋給 Planner Agent。這種回饋方式類

似於 ReAct 框架中的反思（Reflection）過程，確保錯誤診斷信息（例如，操作間依賴性錯誤或服務端邏輯錯誤）能夠被清晰地傳遞。在Re-loop 過程中，MCP 協議扮演了關鍵的上下文連續性角色。所有關鍵的當前狀態資訊（例如，導致失敗的參數序列、錯誤的執行堆疊）必須被安全且一致地維持在 MCP Context 中。這確保了在多步驟操作中，Planner Agent 和 Test Coder Agent 能夠以一致的視角存取所需的診斷資訊，避免狀態遺失。Planner Agent 收到結構化回饋後，啟動 Re-loop 流程，並將修復任務重新委派給 Test Coder Agent。此循環（規劃-執行-分析-修復）賦予了系統強大的強韌性，使其能夠在無需人工干預的情況下，透過反覆迭代優化其行動和計畫。

表 3.2: 範例應對表：處理參數依賴性錯誤

Step	Content	Error & Repair
Executor 執行	Executor Agent 執行腳本，但步驟（2）失敗，返回 404 Not Found。	A2A 傳輸結果：Executor Agent 將 404 狀態碼和堆疊追蹤作為 Artifacts 傳輸給 Analyzer Agent。
Analyzer 分析	Analyzer Agent 透過 LLM 檢查 Manual Document，發現預期結果應為 200 OK。分析結果的 404 狀態碼，與步驟（1）的輸出 user_id 不一致，判斷測試失敗。	MCP Context 標記：Analyzer Agent 將失敗原因標記為：ERROR_TYPE:Inter-operation Dependency Failure，並將此標記和當前 user_id 的值維持在 MCP Context 中。
Planner 重新規劃	Planner Agent 接收到結構化失敗訊息和上下文後，啟動 Re-loop，並重新委派 Test Coder Agent。	Test Coder Agent 修復：代理意識到 user_id 可能在步驟（1）中未被正確提取或格式化。它存取 MCP Context 中的資訊，修改腳本以確保 user_id 被正確解析和注入到步驟（2）的 URL 中，直到測試成功。
生成語義增強規格	FastMCP 服務將 RTSet 寫入規格的 enum 字段。	新的 OpenAPI 規範：id 參數新增 enum:[“U_001234”, “U_987654”, “U_555555”]。
上下文封裝	Planner Agent 將增強後的規格和 Manual Document 封裝。	A2A 狀態資料結構中包含：{API_SPEC_ENHANCED:< 增強後規格 >, EXPECTED_RESULT:“系統應接受...”,...}。

3.5 CI/CD Pipeline GitOps 自動化整合設計流程

本研究階段旨在將測試流程的成功結果自動轉化為持續整合/持續交付（CI/CD）管道中的具體部署動作，關鍵在於實現分散式雙主機架構下的跨協議互操作性，從而消除從測試到部署的人工干預間隙，實現從測試到部署的自動化銜接。本階段的設計核心是依賴 A2A Protocol 來執行遠程、結構化的任務委派，特別適用於分散式且可擴展的代理生態系統。

3.5.1 A2A 跨主機任務委派與職責

本框架利用雙主機設計實現責任隔離，AutoGen Host 一旦測試成功（通過階段三的自校準驗證），Executor Agent 即充當 Client Agent 的角色負責分析意圖並構建一個結構化的 Task 物件，將其發送出去。位於 GitOps Host 上的 GitOps Agent 扮演 Remote Agent(Server) 的角色。Remote Agent 必須發布 Agent Card 來宣告其可用 Skills（例如 Git 操作能力），並執行 Client Agent 委派的任務。這種跨主機的 A2A 實施，展示了協議在互通性方面的優勢。A2A 協定作為點對點任務委派的標準化介面，確保 Client Agent 能夠安全且結構化地與位於不同網路邊界上的 GitOps Agent 協作。此時，A2A 請求扮演了執行訊號（Actuation Signal）的角色，將測試成功的認知結果（來自 AutoGen Host）轉譯為對外部 Git Repository 的實際影響（來自 GitOps Host）。

3.5.2 安全性考量與 GitOps 自動化銜接

本階段的實施必須同時滿足自動化銜接與高安全性的要求。所有核心任務通訊均建議採用 JSON-RPC 或類似的結構化格式，以確保方法調用和參數傳遞的標準化。Client Agent 在發送請求時，必須將測試階段產生的關鍵 Artifacts（例如測試結果、成功狀態）作為 A2A 訊息的一部分傳輸。這些 Artifacts 用於創建可追溯性（Traceability）日誌，並作為 GitOps Agent 執行高權限操作的必要前置條件。在分散式環境下，對高權限的 Git 操作進行存取控制至關重要。GitOps Agent 必須實施嚴格的驗證機制來確認 Client Agent 的身份和訊息完整性（例如利用 JWS（JSON Web Signatures）驗證訊息）。此機制確保了只有經過驗證的自動化流程才能觸發敏感的 Git 操作。GitOps Agent 接收 A2A 結構化請求後，將任務（例如創建 Pull Request 或合併分支）轉化為實際的 Git 操作。此操作是 Execution and Actuation 模組的核心，成功執行後，GitOps Agent 必須將操作結果（例如 Pull Request 連結、新的 Commit ID）作為 Artifacts，透過 A2A 協議回傳給 Client Agent，供最終的報告生成階段（階段五）使用。

3.6 最終報告產出設計

本研究方法論的第五階段為報告生成與結案（Reporting Closure），此階段作為整個複雜多代理流程的最終交付環節，旨在將所有測試活動、協作決策和最終結果轉化為一份完整、結構化且具備可追溯性的最終測試報告。此階段的設計核心在於解決多代理長時間協作下常見的上下文破碎化問題，並確保系統具備完全的透明度（Transparency）與問責性（Accountability）。本階段的理論設計奠基於 MCP 的核心能力，即維持上下文一致性和安全整合工具服務。

Analyzer Agent 或 Planner Agent（作為 Client Agent）將再次呼叫 FastMCP 服務（由 LangChain 程式庫支持的工具互動層），啟動 Reporter Agent。MCP 協定在此處提供了上下文感知層，其核心作用是安全且結構化地存取整個 A2A 流程中累積的上下文數據和執行結果。MCP 在此階段的關鍵任務是確保所有工具和代理在整個複雜流程中上下文的一致性。Reporter Agent 透過 MCP 服務，能夠存取並整合所有 Agent（包括 Planner, Coder, Executor, GitOps Agent）的日誌、A2A 協定中定義的 Artifacts（執行結果）、Manual Document 的預期值以及階段一的前置處理分類結果。這種全面性的數據整合機制，避免了傳統系統中的數據孤島問題，確保最終報告基於連貫且完整的操作歷史。最終生成的報告必須是完整且結構化的。結構化報告是 LLM 驅動的 Agentic 系統中可追溯性的關鍵要素。這使得報告不僅能夠呈現結果，還能提供操作洞察和合規性文件，涵蓋 LLM 的互動、決策依據和協作流程。

本階段的實施將 Reporter Agent 作為一個專門的 Worker Agent，利用 LLM 的語言理解和生成能力來彙整複雜數據並生成結構化輸出：FastMCP 作為資料匯集與服務層 FastMCP 服務應被設計為一個資料匯集層（Data Aggregation Layer），其核心功能是從長短期記憶（STM/LTM）單元中檢索整個協作流程的數據。Reporter Agent 透過 LangChain 抽象化，調用 LLM 來執行報告生成任務。報告的內容以結構化格式（例如 JSON 或 YAML）作為基礎，然後再轉譯為人類可讀的 Markdown 或 HTML 格式。結構化測試報告內容最終報告應包括以下關鍵結構化部分：

1. 測試元數據（Metadata）：測試 ID、OpenAPI 規格版本、測試目標（URL）。

2. 上下文基礎（Context Foundation）：階段一輸出的語義增強後規格的摘要（例如，

RTSet 中使用的關鍵參數值，以驗證測試輸入的品質）。

3. 執行摘要（Execution Summary）：總測試序列數、成功/失敗次數、總執行時間（Task Completion Time, TCT）。

4. 失敗分析與自校準追溯（Failure Analysis & Self-Correction Trace）：對階段三中所有失敗案例的詳細追溯，包括 Analyzer Agent 判定的失敗原因（例如，Inter-operation Dependency Failure）、啟動的 Re-loop 次數，以及最終修復成功的代碼片段。

5. GitOps 行動記錄：記錄階段四中 GitOps Agent 執行的具體操作（例如，Merge Branch 或 Create Pull Request 的連結），以確保測試流程與 CI/CD 流程的可追溯性銜接。

這個階段的設計，確保了整個 LLM 代理測試流程具備持久性和問責性，因為每一個決策和結果都能夠在最終的結構化報告中找到清晰、一致的記錄。整個雙協定整合的流程，從資料輸入到最終報告生成，就像是一個智慧化工廠：A2A 協定是生產線上的工人之間傳遞指令和半成品（任務委派和執行結果），而 MCP 協定則像是中央管理系統，持續記錄所有工作步驟、使用的工具及最終的品質檢測數據，確保最終出廠的產品（測試報告）是完整且可驗證的。

參考文獻

[1] Anu Bajaj and Om Prakash Sangwan. “A Systematic Literature Review of Test Case Prioritization Using Genetic Algorithms”. In: *IEEE Access* 7 (2019), pp. 126355–126375. DOI: 10.1109/ACCESS.2019.2938260.

[2] C. Ebert, D. Bajaj, and M. Weyrich. “Testing Software Systems”. In: *IEEE Software* 39.4 (2022), pp. 8–17. DOI: 10.1109/MS.2022.3166755.

[3] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. “LlamaRestTest: Effective REST API Testing with Small Language Models”. In: *Proceedings of the ACM on Software Engineering* 2.FSE (FSE 2025). To appear in FSE 2025, FSE022.

[4] Tao Zheng et al. “RESTLess: Enhancing State-of-the-Art REST API Fuzzing With LLMs in Cloud Service Computing”. In: *IEEE Transactions on Services Computing* 17.6 (2024), pp. 4225–4238. DOI: 10.1109/TSC.2024.3489441.

[5] Vu Nguyen, Tien N. Nguyen, et al. “KAT: Dependency-Aware Automated API Testing with Large Language Models”. In: *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Authors listed in [5]. 2024, pp. 82–92. DOI: 10.1109/ICST60714.2024.00017.

[6] Chunhui Wang et al. “Automatic Generation of Acceptance Test Cases From Use Case Specifications: An NLP-Based Approach”. In: *IEEE Transactions on Software Engineering* 48.2 (2022), pp. 585–616. DOI: 10.1109/TSE.2020.2998503.

[7] J. C. Alonso et al. “ARTE: Automated Generation of Realistic Test Inputs for Web APIs”. In: *IEEE Transactions on Software Engineering* 49.1 (2023), pp. 348–363. DOI: 10.1109/TSE.2022.3150618.

[8] U. M. Borghoff, P. Bottoni, and R. Pareschi. “Beyond Prompt Chaining: The TB-CSPN Architecture for Agentic AI”. In: *Future Internet* 17.8 (2025), p. 363. DOI: 10.3390/fi17080363.

[9] J. Wang et al. “Software Testing With Large Language Models: Survey, Landscape, and Vision”. In: *IEEE Transactions on Software Engineering* 50.4 (2024), pp. 911–936. DOI: 10.1109/TSE.2024.3368208.

[10] C. Xu et al. “FlexFL: Flexible and Effective Fault Localization With Open-Source Large Language Models”. In: *IEEE Transactions on Software Engineering* 51.5 (2025), pp. 1455–1471. DOI: 10.1109/TSE.2025.3553363.

[11] A. Bandi et al. “The Rise of Agentic AI: A Review of Definitions, Frameworks, Architectures, Applications, Evaluation Metrics, and Challenges”. In: *Future Internet* 17.9 (2025), p. 404. DOI: 10.3390/fi17090404.

[12] X. Hou et al. “Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions”. In: *arXiv preprint arXiv:2503.23278* (2025). URL: https://arxiv.org/abs/2503.23278.

[13] S. Fatima, T. A. Ghaleb, and L. Briand. “Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests”. In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 1912–1927. DOI: 10.1109/TSE.2022.3192008.

[14] A. Ehtesham et al. “A Survey of Agent Interoperability Protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP)”. In: *arXiv preprint arXiv:2505.02279* (2025). Based on details in [1], [2], likely a recent survey or preprint. URL: https://arxiv.org/abs/2505.02279.

[15] K. Li and Y. Yuan. “Large Language Models as Test Case Generators: Performance Evaluation and Enhancement”. In: *arXiv preprint arXiv:2404.13340* (2024). URL: https://arxiv.org/abs/2404.13340.

[16] E. T. Barr et al. “The Oracle Problem in Software Testing: A Survey”. In: *Survey or Review Journal* 41.5 (2015). Full author/publication details not available in excerpt [10]., pp. 507–525. DOI: 10.1109/TSE.2014.2372785.

[17] T. Şimşek, A. Ç. Gülşen, and G. A. Olcay. “The Future of Software Development With GenAI: Evolving Roles of Software Personas”. In: *IEEE Engineering Management Review* 53.4 (2025). DOI confirms journal and dates [8]., pp. 34–45. DOI: 10.1109/EMR.2024.3454112.

[18] Alejandro Sánchez Guinea, Grégory Nain, and Yves Le Traon. “A systematic review on the engineering of software for ubiquitous systems”. In: *The Journal of Systems and Software* 118 (2016), pp. 251–276. DOI: 10.1016/j.jss.2016.05.024.

[19] C.-A. Sun et al. “Detecting Inconsistencies in Microservice-Based Systems: An Annotation-Assisted Scenario-Oriented Approach”. In: *IEEE Transactions on Services Computing* 17.5 (2024), pp. 2194–2209. DOI: 10.1109/TSC.2024.3399652.

[20] M. Zhang and A. Arcuri. “Open Problems in Fuzzing RESTful APIs: A Comparison of Tools”. In: *ACM Transactions on Software Engineering and Methodology* 33.1 (2024), 27:1–27:41. DOI: 10.1145/3597205.

[21] S. Karlsson, A. Čaušević, and D. Sundmark. “QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs”. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, pp. 131–141. DOI: 10.1109/ICST46399.2020.00023.

[22] O. Banias et al. “Automated Specification-Based Testing of REST APIs”. In: *Sensors* 21.8 (2021), p. 2699. DOI: 10.3390/s21082699.

[23] S. Hosseini and H. Seilani. “The role of agentic AI in shaping a smart future: A systematic review”. In: *Array* 26 (2025), p. 100399. DOI: 10.1016/j.array.2025.100399.

[24] Vaibhav Tupe and Shrinath Thube. “Demonstrating Multi-Agent Collaboration via Agent-to-Agent and Model Context Protocols: An IT Incident Response Case Study”. In: *2025 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. Project Demo: https://youtu.be/2YCs6Axgifs. 2025, pp. 1–5. DOI: 10.1109/SOSE67019.2025.00013.